19 | 综合案例：电商平台技术架构是如何演变的？

2020-04-03 王庆友

架构实战案例解析

进入课程 >

讲述：王庆友

时长 11:25　大小 10.47M

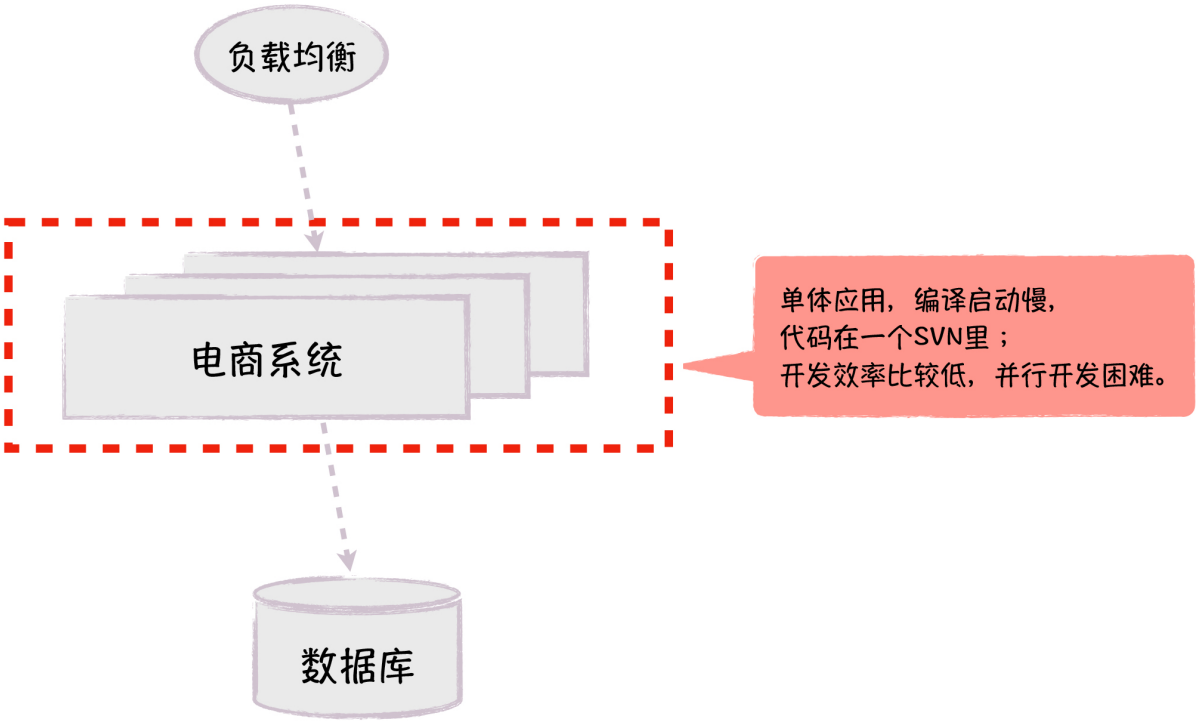
你好，我是王庆友。

在前面的几讲中，我分别和你介绍了技术架构的高可用、高性能、可伸缩等目标，并通过实际的案例说明了如何实现这些目标。今天呢，我会通过一个综合案例，来说明电商平台的技术架构是如何演变的，让你可以全面地理解如何实现这些目标。

一个实际的电商系统很复杂，在案例介绍中，为了简化，我用比较有代表性的**交易系统**和**账户系统**来代表整体的电商系统，并具体分析这两个系统在电商平台发展过程中，它们[illegible]了什么瓶颈，以及我们在技术架构上是如何解决的。

这一讲会包含很多架构图，每一张图都代表了不同时期的架构设计，为了方便你更好地理解它们，在每张架构图中，我都用红色方框圈出了当前架构存在的问题，用绿色实体部分代表了上一个架构所存在问题的解决办法，希望你听完今天的讲解，能够结合这些架构图，加深对技术架构的理解。

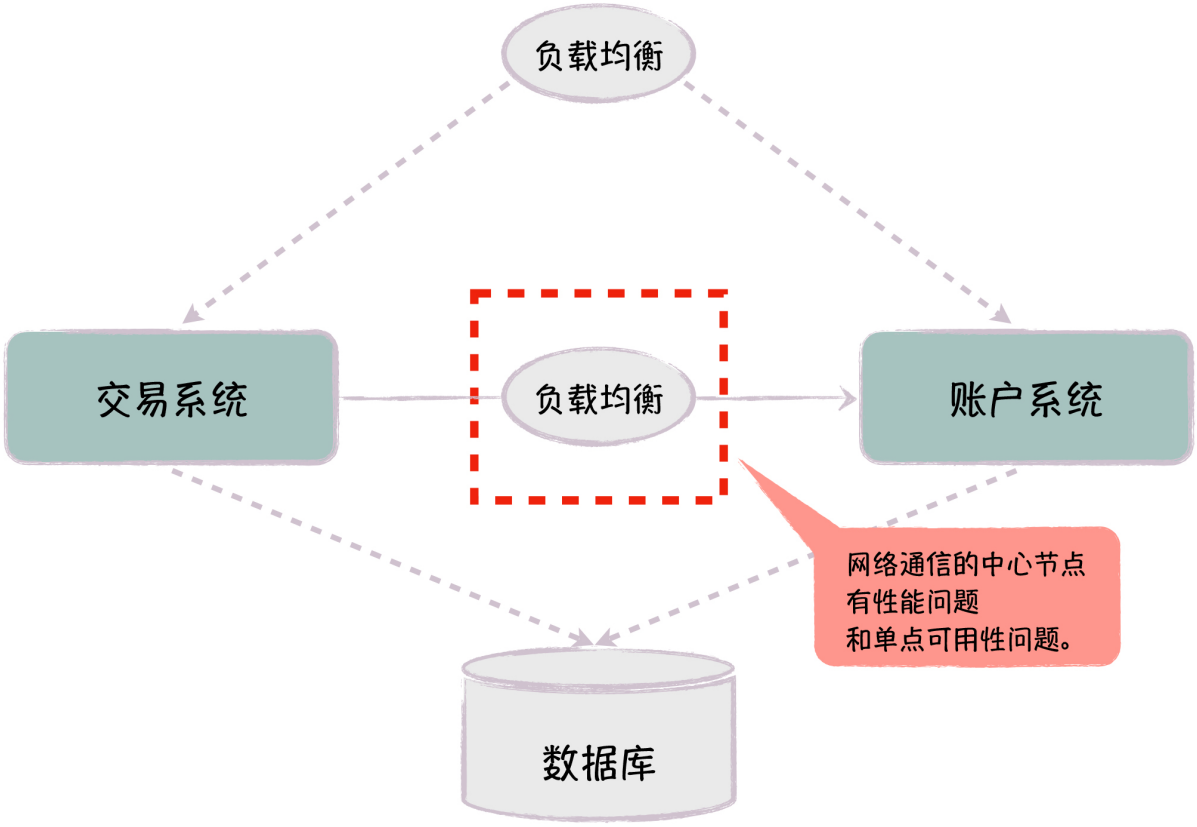
单体系统



第一代的电商系统是一个单体架构，所有的代码都打包在一个应用里，部署的时候会有多个实例，我们通过**负载均衡**，把用户请求分发到具体的实例中。这个时候，所有的数据表还在一个数据库里。

这里的问题是，单体应用的所有代码都放在一起，代码编译需要很长时间，应用启动也需要很长时间，并且代码相互依赖，开发效率低，并行开发困难。随着单体应用的体量越变越大，这些问题也越来越突出。

SOA 架构

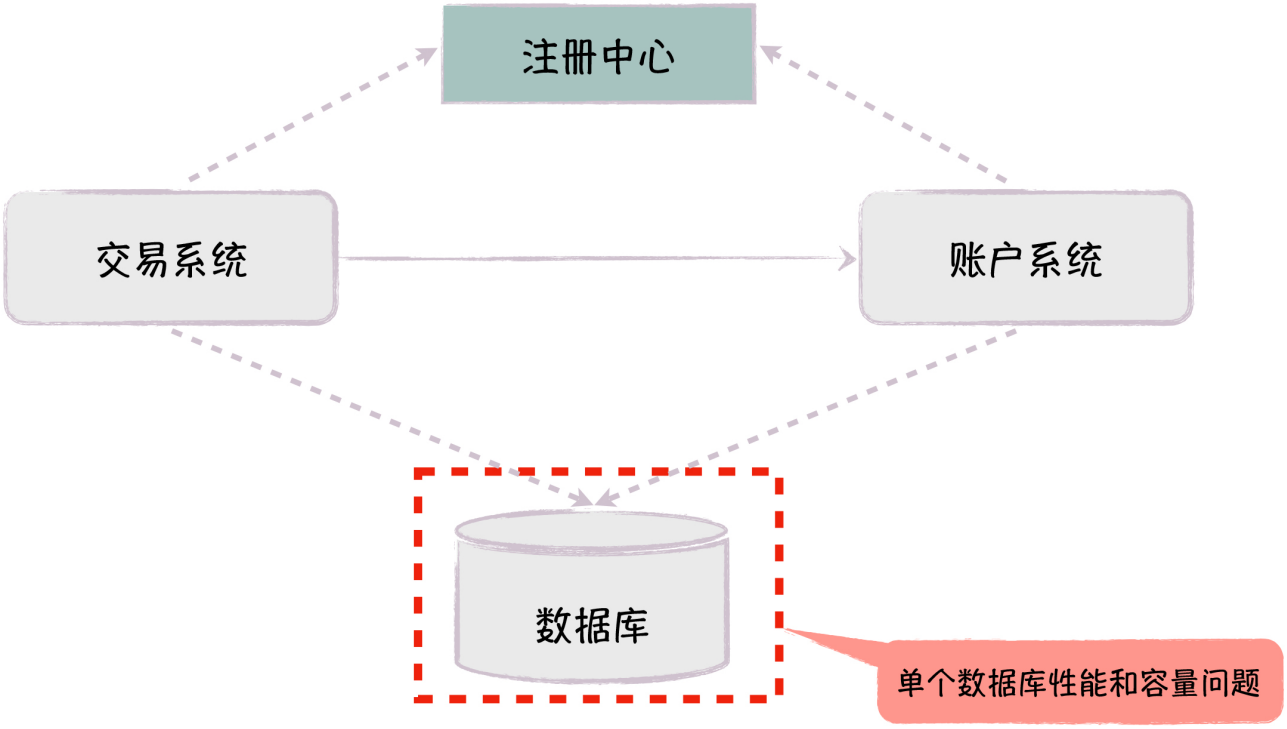


针对单体应用体量过大带来的问题，我们**对系统进行拆分**，把整体系统分为多个子系统。比如在图中，我们把系统拆分为交易系统和账户系统，这两个系统（或者说服务）通过内部的负载均衡进行相互调用，这个时候，底层数据库还没有拆分，两个系统还是访问同一个数据库。

通过拆分，系统整体就变成了 SOA 架构，这样，我们减少了代码冲突，系统的开发也变得更加高效，部署的时候，我们也更容易针对各个系统的处理能力进行水平扩展。

但这里的问题是，内部服务通过中心化的负载均衡进行访问，中心化的负载均衡增加了服务的调用时间。此外，在电商场景下，内部的服务很多，服务调用的频率很高，每秒可能有上百万次，导致了负载均衡的连接能力不够。而且负载均衡是单点，如果它出了问题，很容易引发系统整体的可用性问题（即使负载均衡是多实例，当系统流量很大时，也会因为某台负载有问题，导致其他节点压力增大而引起雪崩效应）。

服务调用去中心化



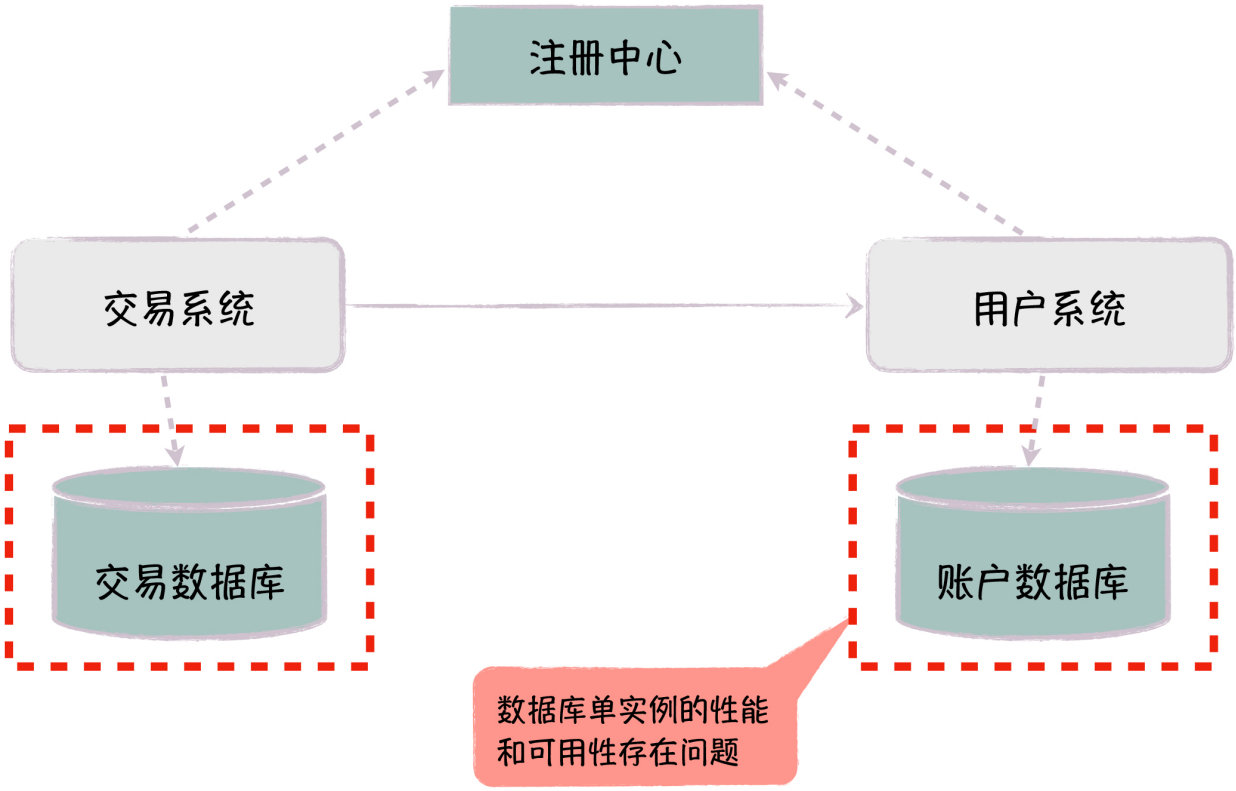
针对内部服务路由中心化的问题，我们去掉了内部的负载均衡，加入了**服务注册中心**，比如 ZooKeeper。

当服务实例启动或退出时，它们会自动在注册中心进行注册或销毁，服务的客户端和注册中心保持长连接，可以实时地获取可用的服务列表；然后在客户端，根据相应的算法选择服务实例，直接调用服务。每次调用无需经过注册中心，如果注册中心有问题，也只是新的服务实例无法注册，或者是已有的服务实例无法注销，这对客户端调用服务的影响是非常有限的。

当然，通过注册中心和更体系化的微服务框架，我们还可以实现完善的**服务治理**，包括服务隔离、熔断、降级，这些都是对原来简单的负载均衡方式的加强，能够进一步提升服务的可用性。

现在，我们解决了服务调用的问题，但随着业务量逐渐变大，数据表越来越多，数据量也越来越大，**单个数据库（比如 Oracle）的性能和储存容量已经无法满足需求了**。这个时候，我们就需要对数据库进行改造，提升它的处理能力。

垂直分库

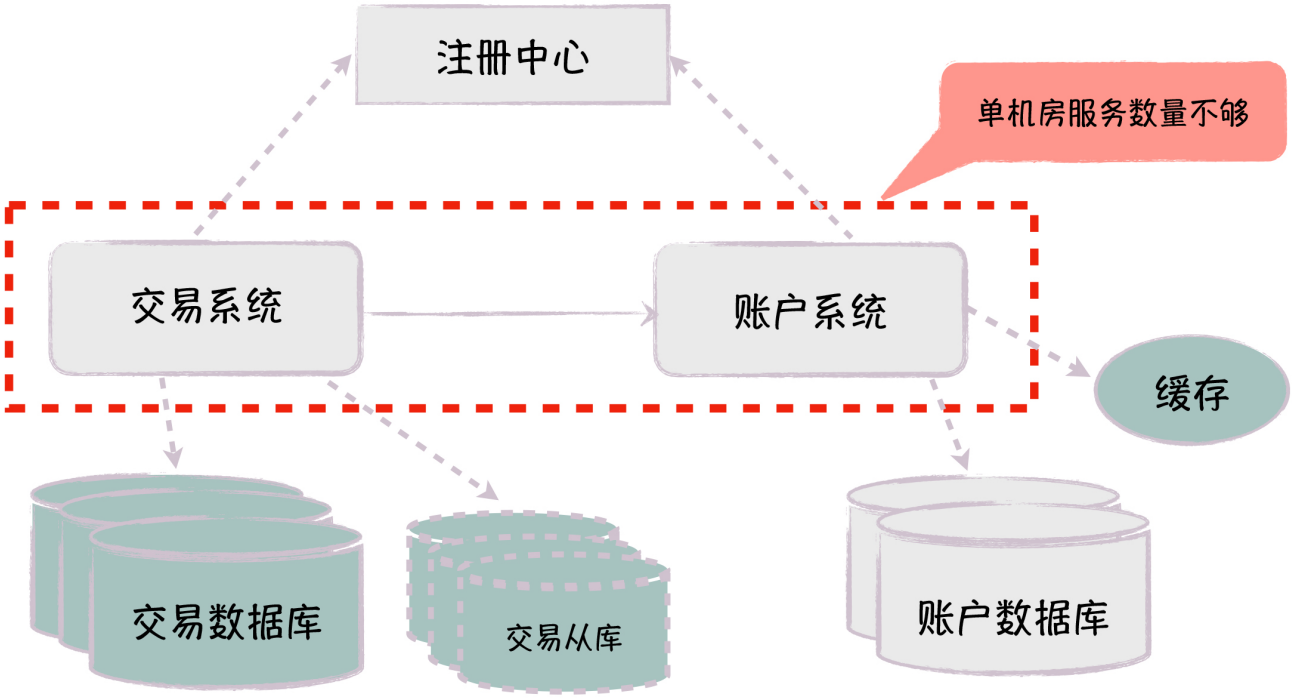


对于单个数据库性能和容量瓶颈，解决的办法就是，我们对数据库进行**垂直拆分**，按照业务拆分为交易数据库和账户数据库，这样就可以满足它们各自的容量和性能需求，同时也避免了不同业务数据表之间的相互耦合。

你可以认为垂直分库是系统拆分的第二阶段，这样，通过第一阶段的应用代码拆分和这里的数据库表拆分，交易系统和账户系统就可以独立发展。

不过，新的问题又来了，垂直分库后，每个数据库都是单实例。随着业务的发展，和原来系统只有单个数据库类似，现在交易系统也只有一个数据库，它的性能和容量还是有问题，并且数据库单实例也带来了可用性的问题，如果数据库挂了，相应的系统也就不可用。

水平分库及高可用部署



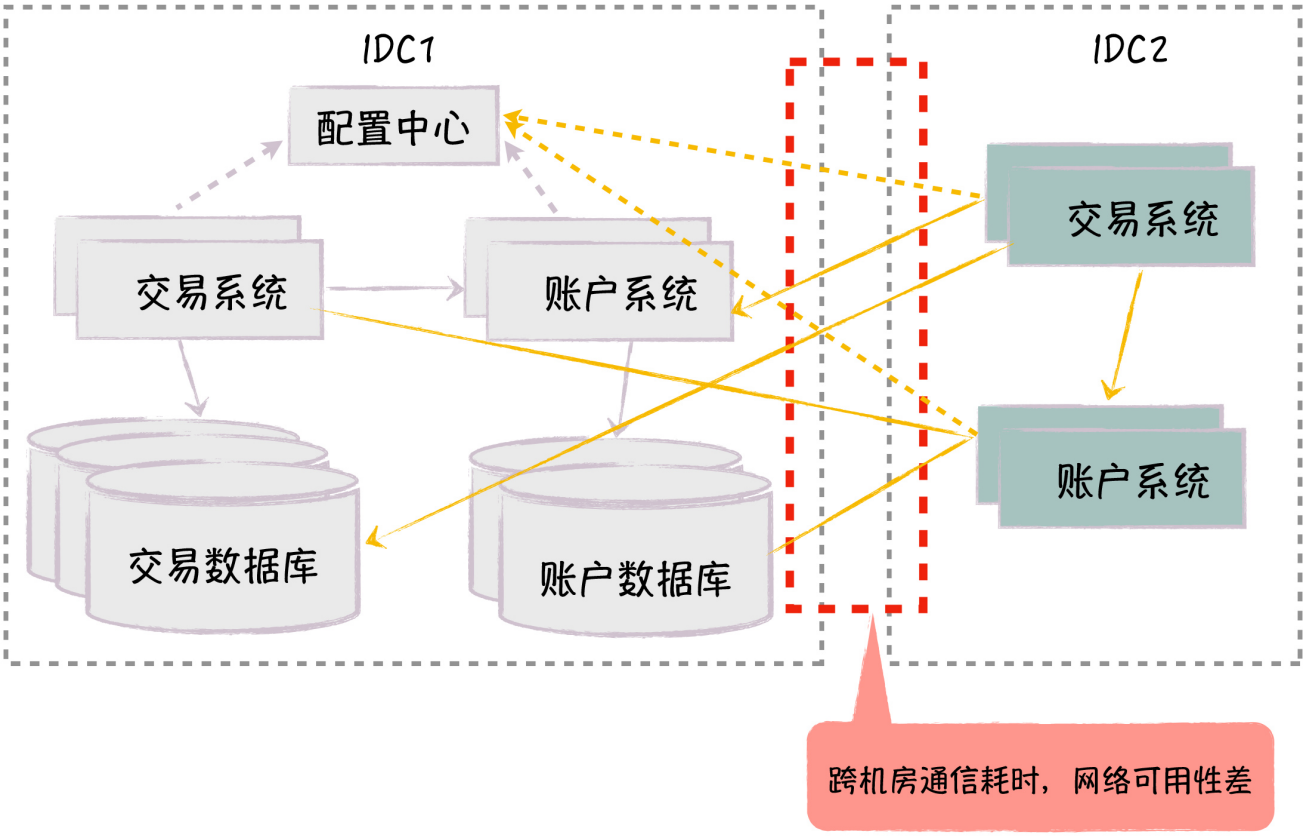
针对单个数据库的可用性问题，我们可以采用 **MHA 高可用**（Master High Availability）方式部署。比如数据库部署一主多从，通过 MHA 机制，我们可以实时检测主库的可用性，如果主库有问题，系统会自动 Failover（故障转移）到最新的从库。另一方面，我们还可以利用多个从库支持**读写分离**，减轻主库的访问压力。

针对单个数据库的性能和容量问题，首先我们可以**引入缓存**，在高读写比的场景下，让应用先访问缓存，大大减轻对底层数据库的压力。然后，我们可以对数据库按照某个维度（比如用户维度），进行**水平拆分**，把数据记录分布到多个实例中，最终分散主库的写压力以及数据存储的瓶颈（在上一讲中，我已经具体介绍过了，你可以点击链接去回顾内容）。

在实践中，我们还可以提供**多套水平分库**。比如说，针对交易数据，我们可以同时按照用户维度和商户维度进行水平分库，用户维度的库用于前台用户下单的场景，商户维度的库用于后台商家履单的场景。这里，只有用户维度的分库会支持**写**，我们通过数据同步机制，把用户维度分库的更新同步到商户维度的分库里。

当系统体量发展到了一定程度，我们又碰到了新的问题：单个机房的服务器不够用，无法在同一个机房找到更多的机器部署交易系统和账户系统。

多机房部署

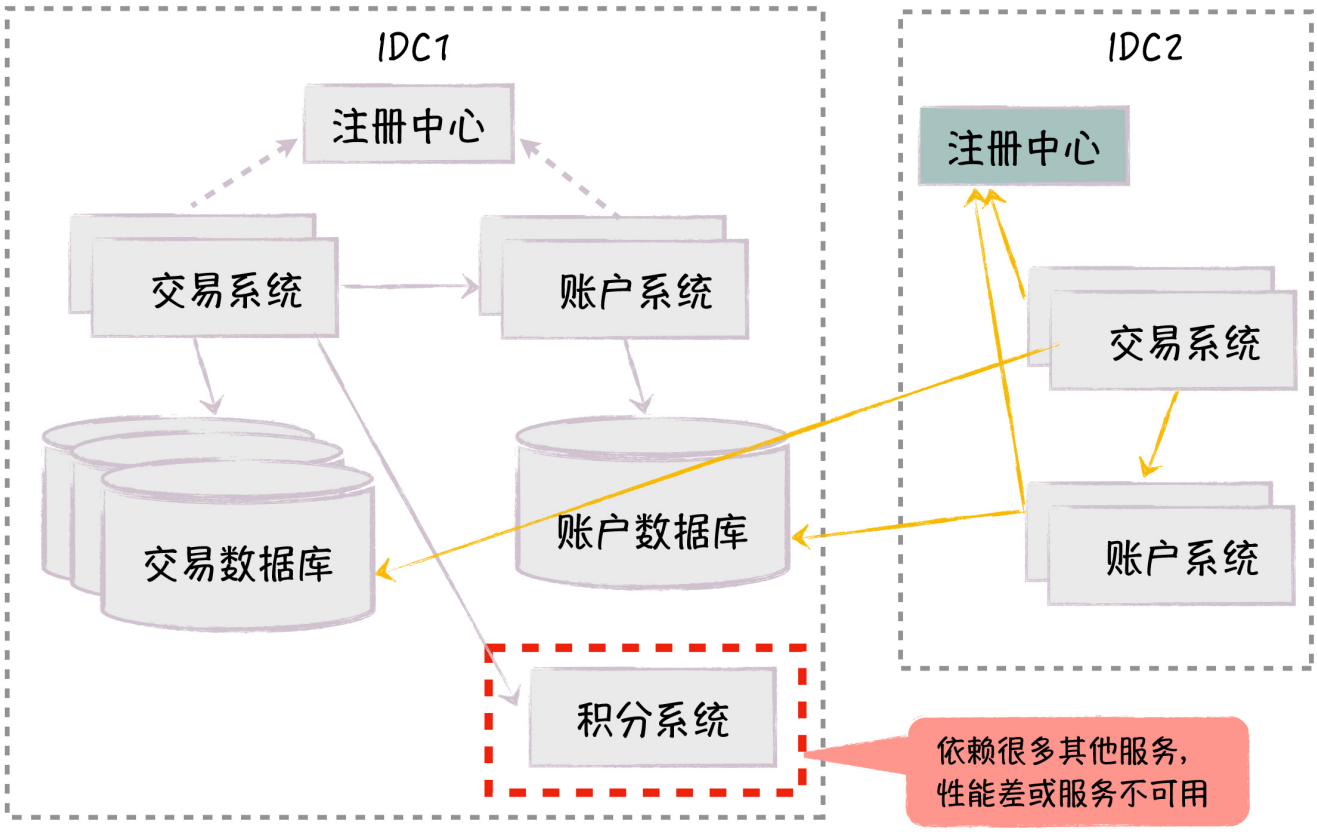


对于单机房服务器不够的问题，我们可以在新的机房部署交易系统和账户系统，为了落地方便，所有服务还是注册到旧机房的注册中心，数据还是存放在旧机房的交易数据库和账户数据库。 这样，我们通过在新机房部署应用，对应用节点进行水平扩展，从而解决了单机房机器不足的问题。

但这里产生了跨机房访问的问题：首先，我们只有一个服务注册中心，服务实例一部分部署在老机房，一部分部署在新机房，对于服务调用者来说，它会同时访问新旧机房的服务实例；其次，数据库部署在老机房，新机房的应用会访问旧机房的数据库。

这两种情况，都会产生大量的跨机房访问，我们知道，根据机房物理距离的不同，跨机房访问的网络延时在数十毫秒到数百毫秒之间，是机房内部通信耗时的上千倍，这会对**应用的性能**产生很大影响，而且跨机房的**网络可用性**也经常是一个问题。

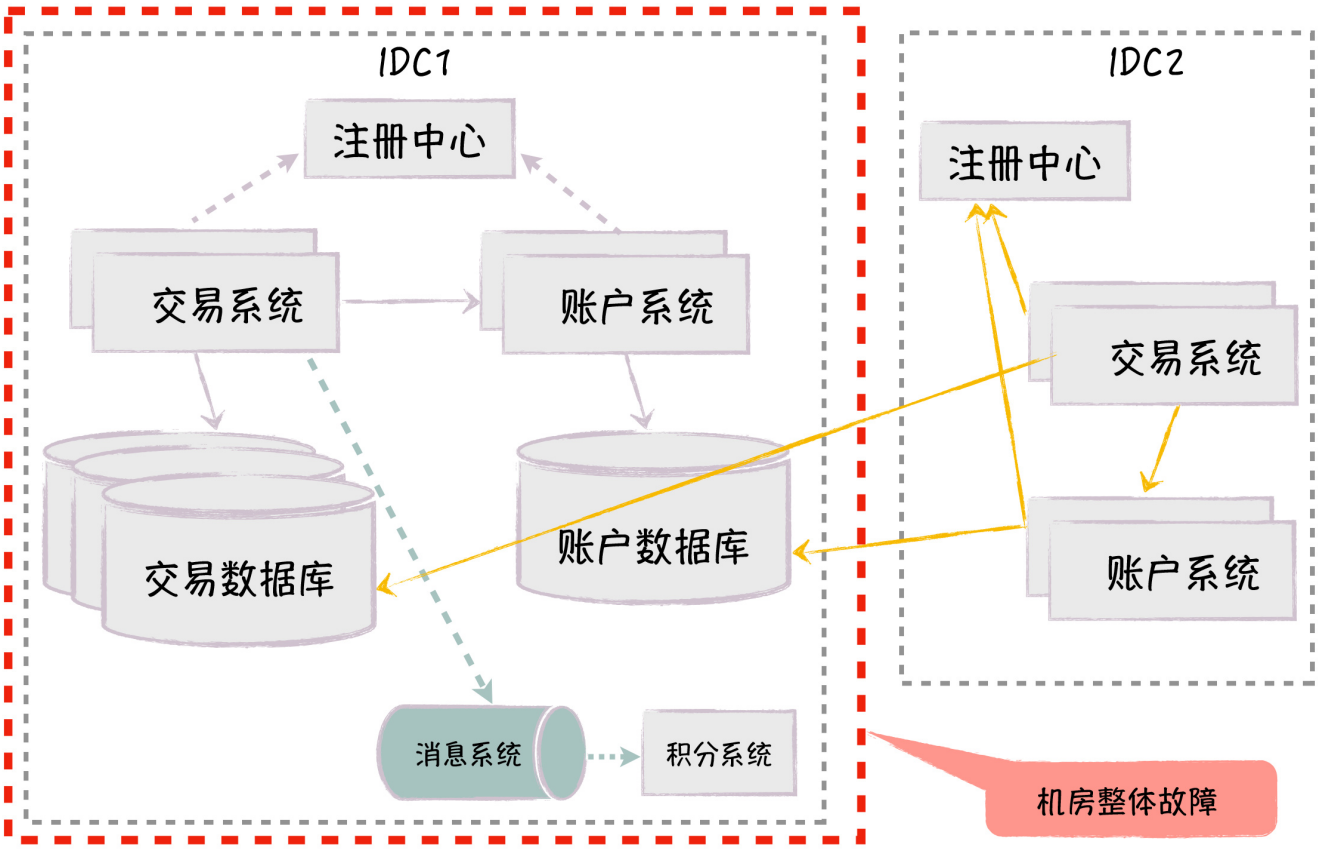
服务调用本地化



为了避免服务的跨机房访问，我们**在新机房也单独部署了服务注册中心**，让每个机房的服务注册到同机房的注册中心。这样，客户端的服务调用会路由到同机房的服务端，实现了服务调用的本地化，大大降低了跨机房通信带来的延时和不可用性问题。

这时，**随着业务越来越复杂，新的问题又来了**：交易系统会依赖很多周边服务。比如下单后，我们需要给用户送积分，交易系统会同步调用积分服务。但是同步调用积分服务，一方面会影响下单的性能，另一方面如果积分服务不可用，会导致核心的下单功能失败。

依赖分级管理

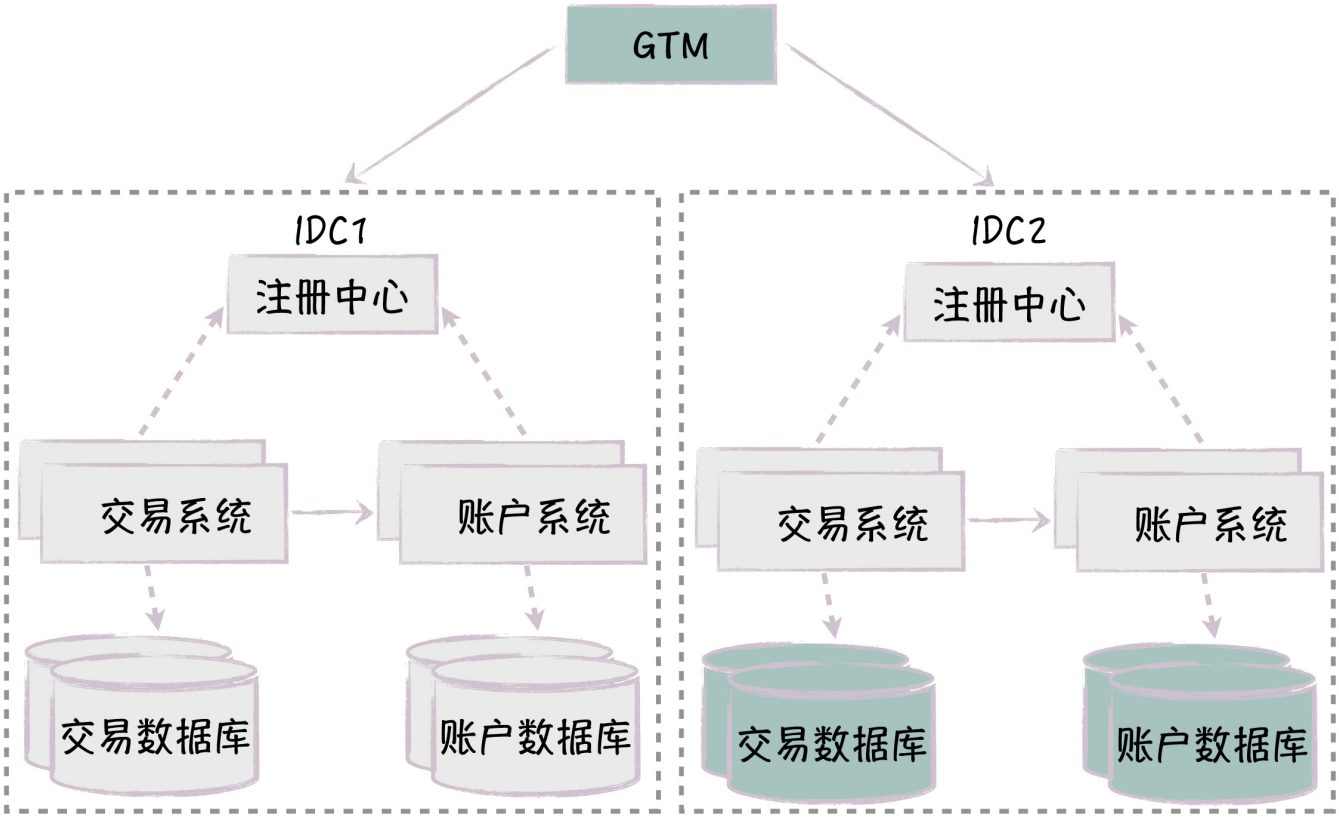


对于外部服务依赖的可用性问题，我们的解决办法是，针对这些外部依赖进行**分级管理**，根据依赖功能的重要性不同，把它们分为强依赖和弱依赖。

> 对于强依赖，我们**实时同步调用**，比如在用户下单时调用库存服务，由于库存非常重要，必须实时扣减，如果调用库存服务失败，下单也失败。
>
> 对于大量的弱依赖，我们以**异步消息**的方式进行信息同步，比如对于积分服务，可以通过柔性事务来保证数据的最终一致性，这样大大提升了核心系统的性能和可用性。

不过，这里存在的问题是，新机房的交易系统和账户系统都在访问老机房的数据库，有跨机房数据库访问的性能问题，以及老机房整体故障带来的可用性问题。比如说，机房断电，通信光纤有问题或者发生自然灾害，导致老机房整体不可用，这就会导致所有系统都不可用。

多机房独立部署



针对机房整体不可用的问题，解决方案是，我们**在多个机房做对等的部署**，这样每个机房的系统可以形成内部闭环，包括服务、注册中心和数据库，机房之间不产生直接的相互依赖，从而实现了机房级别的水平部署。

如果系统的单元化做得完善，我们还可以进一步支持**虚拟机房**的概念，一个物理机房可以部署多个虚拟机房，每个虚拟机房包含了一个完整的系统。通过多机房独立部署，我们极大地提升了系统的可用性、处理能力和可伸缩性，可以应对系统面临的各种异常情况。

另外，最近几年，容器化技术的发展很快，原来很多的电商平台都是基于**虚拟机**部署，现在也纷纷改造为用 **Docker+K8s** 的方式部署，这大大提升了资源的利用率、系统的弹性伸缩能力。在面临资源瓶颈时，你可以考虑用这种方式来优化系统的部署。

总结

今天，我基于一个简化的电商系统模型，与你分享了电商平台的技术架构发展过程，我们是如何通过一步步的架构升级，解决系统各个阶段出现的高可用、高性能和可伸缩问题的，相信你现在对技术架构如何应对各种系统性挑战，有了更深入的认识。

值得注意的是，系统的技术架构变化不一定要完全遵循这个过程，不同的业务、不同的发展阶段，对系统的要求都是不一样的，这里我给出的只是典型的问题和解决手段，希望你在工

作中，能够具体情况具体分析，灵活地运用这些手段。

业务在不断发展，新的问题会不断出现，但技术也在不断地进步，解决的手段层出不穷，我们需要不断学习，找到新的手段来解决问题。

最后，给你留一道思考题：你的公司当前的系统架构处于哪个阶段，面临什么样的问题呢？

欢迎在留言区和我互动，我会第一时间给你反馈。如果觉得有收获，也欢迎你把这篇文章分享给你的朋友。感谢阅读，我们下期再见。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　18 | 可伸缩架构案例：数据太多，如何无限扩展你的数据库？

下一篇　20 | 从务实的角度，给你架构设计的重点知识和学习路径

精选留言 (6)

写留言

Jeff.Smile
2020-04-03

这篇特干货，👏。

尤其是文章开头的使用注册中心代替传统nginx.降低调用耗时的方式，突然间明白了很多。这应该就是为什么分布式架构流行的原因之一了。感谢！

1

芒果
2020-04-03

老师您好，看完文章有2个问题？
1.请问多机房独立部署的话，数据同步这块是怎么做的呢？
2.我们公司现在也在采用k8s和docker的方式部署应用，但是类似数据库还是没有容器化。请问目前业界也是这样吗？还是已经实现全部上k8s部署了？

展开

作者回复: 1. 理想情况下，各个机房业务自成闭环，比如A机房处理用户Id是2倍数的用户交易，B机房处理另外用户。这里的数据库是水平分库，这两部分用户的订单放在不同的库里，A机房的订单会同步给B机房，反之一样。每个机房都会有全量数据。如果机房靠的近，可以采用数据库自带的主从同步方式，隔得远，使用程序同步机制。
2. 有状态的数据库一般不用docker部署。

2 1

小洛
2020-04-05

这篇是一篇很好的体系化梳理文章 谢谢老师的分享，请教您一个问题
1、如果是多机房独立部署，假设有北方机房，南方机房，这个距离很远（甚至可以是国内、国外这么远），那么北方的用户来到南方使用我们的系统，我们是路由回北方机房进行服务提供吗？要如何优化这种业务场景呢？

展开

作者回复: 原则上来说，北方的用户来到南方使用我们的系统，我们是路由回北方机房，这个只是接入层长途路由，服务端内部还是走北方机房，性能问题不大。

川杰
2020-04-03

接上一个。
我可能没有阐述清楚，我们有一张大的结算指令表，表中有一个交易编号的字段，如果结算服务不访问交易，那么只能:
1.调用结算服务，返回要计算的结算指令上的所有交易编号。

2.调用交易服务，获取交易数据。...

展开

作者回复: 你这里交易服务和结算服务有上下游关系，有直接调用关系也正常。
或者你拆分的更彻底一些，把交易服务分为偏上层的和偏底层的，偏上层的交易服务同时调用底层交易服务和结算服务。

2

川杰
2020-04-03

老师好，请教一个问题。
我们系统目前是单体应用，正在往服务化做拆分，目前的计划是先拆分一个跟主业务关系不大的模块（结算模块）。
现在碰到了一个问题，因为我们是金融系统，有交易模块，结算模块等。当做了服务化拆分后，结算模块和交易模块被分离。...

展开

hello
2020-04-03

老师，请教您一个问题，您说的多机房独立部署方案，那存储问题是如何解决的?

作者回复: 可以参考我另一个问题的回复